컴파일러 과제1 보고서 (보충)

20181755 이건희

2024년 9월

메일을 확인하고 급하게 다시 작성하였습니다.

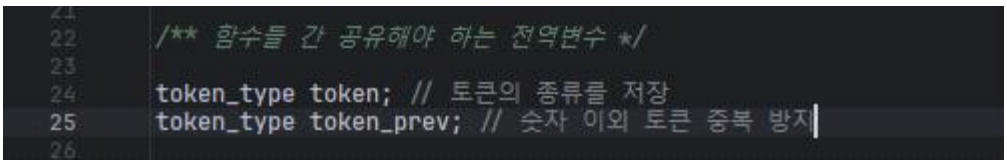
2. 코드 설명

강의교안에서 과제 프로그램의 수도코드는 어느정도 주어졌고, 과제로 해야 할 것은 사실상 get_token 함수를 완성하는 것 뿐입니다. get_token 함수는 수도코드의 주석에 의하면 1. 다음 토큰을 토큰에 저장, 2.숫자를 읽어 num에 저장하는 두 가지의 기능을 해야 하고, 처음에는 당연히 이들만 생각하며 코드를 완성했습니다. 일단 문법적으로 올바른 수식들을 넣어보니 전부 올바르게 계산이 되었습니다. 이전의 보고서에도 언급했듯 시간이 모자랐던 관계로 되는 경우보다 안되는 경우를 테스트 하는 것이 맞을 것 같아 문법적으로 올바르지 못한 수식들을 넣어보았는데, 이들이 에러가 없이 잘못된 정답을 출력하는 경우가 꽤나 많아서 생각이 깊어졌습니다. gcc와 같은 실제 컴파일러를 생각해보면 문법적으로 올바르지 않은 문장에 대하여 에러를 발생시켜야 하는데, 본 과목은 컴파일러이기에 올바른 문장만 넣어 올바르게 작동되는 것을 확인하는 것은 의미가 없다 생각했습니다. 틀린 문장에 대해 에러를 발생시키기 위해 여러 틀린 케이스들을 생각해보고, 이를 처리하게끔 수정을 했습니다.

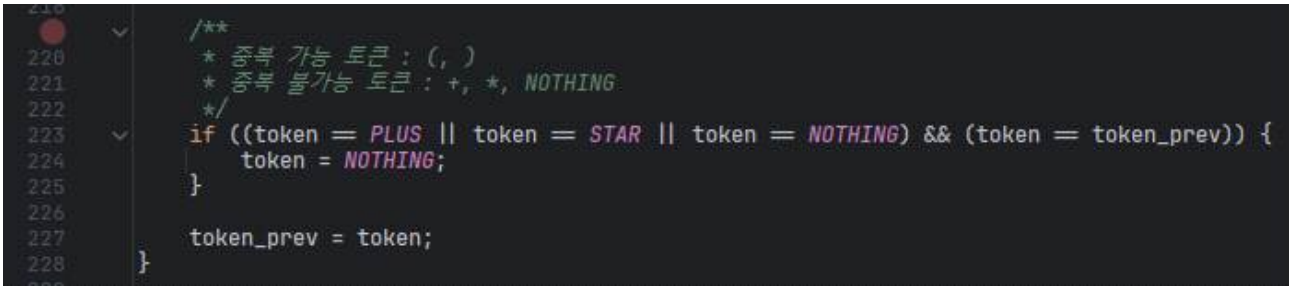
먼저 문제가 되었던 것은 공백의 처리입니다. 수도코드에 살만 붙힌 초기 코드의 테스트 시 “1+2*3”과 같은 정제된 입력만을 넣었습니다. 하지만 예를 들어 “ 8 + 7 ”(오타 아님)이라는 스트링이 주어진다 하면, 각 숫자의 좌우로 있는 공백은 NOTHING으로 처리해야 합니다. 이를 테스트해보니 문법적으로 올바른 수식인데도 공백을 전혀 걸러주지 못하고 에러가 발생했습니다.

이를 해결하기 위하여 substring의 시작을 가리키는 포인터를 공백이 아닌 곳까지 움직이게끔 하여 아래의 enum token_type 1~6을 판단하는 if-else문에는 무조건 공백이 아닌 문자가 주어지게끔 수정했습니다.

다음으로 문제가 되었던 것은 연속된 token의 처리입니다. 생각을 해보면, LPAREM, RPAREM은 충분히 중복될 수 있습니다. "(((3+4)*7)+6)*5"와 같이 괄호가 중복되어도 해당 수식은 올바른 수식입니다. 하지만 PLUS, STAR는 중복이 불가능합니다. "1++2", "3**4"와 같이 연산자가 중복된 수식을 올바른 수식이 아닙니다.

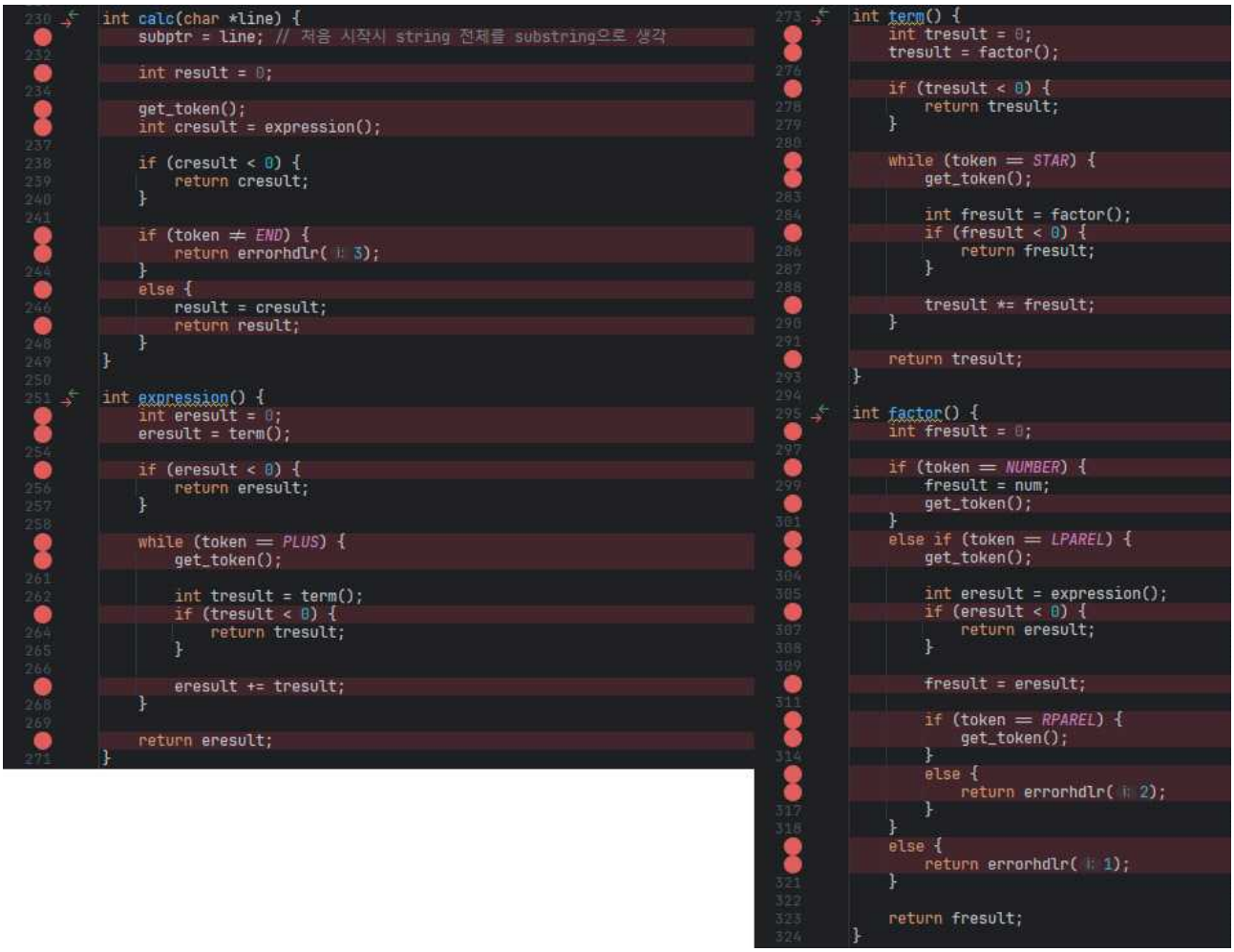


이를 해결하기 위하여 이전 토큰과 현재 토큰 비교를 위한 전역변수를 하나 설정했고



get_token()의 마지막에 중복이 불가능한 토큰이 중복되었을 경우 이를 NOTHING으로 두어 에러 -1을 발생시키도록 수정했습니다.

마지막으로 문제가 되었던 것은 calc()(수도코드의 main()에 해당), expression(), term(), factor()가 함수들을 서로 재귀적으로 호출하는데 같은 전역변수인 token_type token, token_prev를 접근하다보니 문법상 맞지 않는 수식을 대입하면 원하지 않는 분기로 빠지는 것이였습니다. 본 프로그램은 덧샘/곱샘 연산만 지원하며, 다뤄질 수도 무조건 0을 초과하기에 연산의 결과는 무조건 0 이상이고, 따라서 에러코드를 음의 integer로 처리하는 것은 타당합니다. 에러처리로 빠지는 분기는 수도코드에서 안건드렸기에 에러코드가 -1~-3 사이에서 나와야 하는데, -90, -7과 같이 말도 안되게 나오는 경우가 있었습니다. 오류가 났을 시 오류값을 return 하는 것이 아닌 연산하는데 있어서 변수로 사용했다는 의미입니다.



```c
int calc(char *line) {
    subptr = line; // 처음 시작시 string 전체를 substring으로 생각

    int result = 0;

    get_token();
    int cresult = expression();

    if (cresult < 0) {
        return cresult;
    }

    if (token != END) {
        return errorhdlr( n: 3);
    }
    else {
        result = cresult;
        return result;
    }
}

int expression() {
    int eresult = 0;
    eresult = term();

    if (eresult < 0) {
        return eresult;
    }

    while (token == PLUS) {
        get_token();

        int tresult = term();
        if (tresult < 0) {
            return tresult;
        }

        eresult += tresult;
    }

    return eresult;
}
```

```c
int term() {
    int tresult = 0;
    tresult = factor();

    if (tresult < 0) {
        return tresult;
    }

    while (token == STAR) {
        get_token();

        int fresult = factor();
        if (fresult < 0) {
            return fresult;
        }

        tresult *= fresult;
    }

    return tresult;
}

int factor() {
    int fresult = 0;

    if (token == NUMBER) {
        fresult = num;
        get_token();
    }
    else if (token == LPAREL) {
        get_token();

        int eresult = expression();
        if (eresult < 0) {
            return eresult;
        }

        fresult = eresult;

        if (token == RPAREL) {
            get_token();
        }
        else {
            return errorhdlr( n: 2);
        }
    }
    else {
        return errorhdlr( n: 1);
    }

    return fresult;
}
```

이를 해결하기 위하여 각 함수의 모든 분기에 브레이크포인트를 걸어서 디버그하며 한 단계씩 면밀히 살펴보고 수정하여 에러코드가 연산에 섞여 들어가지 않도록 원천적으로 봉쇄했습니다.

올바른 케이스가 작동하는 것을 보여주는 것만 생각한다면 쉬운 과제였으나, 틀린 케이스가 작동하지 않는 것을 보여주는 것까지 생각한다면 조금은 까다로운 과제가 아니였나 싶습니다.